Lifting Micro-Update Models from RTL for Formal Security Analysis

Adwait Godbole (✉)  
adwait@berkeley.edu  
University of California, Berkeley  
Berkeley, CA, USA

Yatin A. Manerkar  
manerkar@umich.edu  
University of Michigan  
Ann Arbor, MI, USA

Sanjit A. Seshia  
sesha@berkeley.edu  
University of California, Berkeley  
Berkeley, CA, USA

Kevin Cheang ✉  
kcheang@berkeley.edu  
University of California, Berkeley  
Berkeley, CA, USA

Abstract
Hardware execution attacks exploit subtle microarchitectural interactions to leak secret data. While checking programs for the existence of such attacks is essential, verification of software against the full hardware implementation does not scale. Verification using abstract formal models of the hardware can help provide strong security guarantees while leveraging abstraction to achieve scalability. However, hand-writing accurate abstract models is tedious and error-prone. Hence, we need techniques to generate models that enable sound yet scalable security analysis automatically.

In this work, we propose micro-update models as a modelling framework that enables sound and abstract modelling of microarchitectural features. We also develop algorithms to generate micro-update models from RTL semi-automatically. We implement our modelling and generation framework in a prototype tool called PAUL. We evaluate our approach by synthesizing micro-update models for the Sodor5Stage processor and components from the cva6 (Ariane) processor. We demonstrate how these models can be generated hierarchically, thus increasing scalability to larger designs. We observe up to 8× improvement in run time when performing analysis with the generated models as compared to the source RTL.

CCS Concepts  • Security and privacy → Hardware reverse engineering; Side-channel analysis and countermeasures; Formal security models; Logic and verification; • Hardware → Functional verification; • Theory of computation → Verification by model checking; Models of computation; • Software and its engineering → Formal methods;

ACM Reference format:
https://doi.org/10.1145/3620665.3640418

1 Introduction
The landscape of hardware execution attacks has evolved since Spectre [44] and Meltdown [46]. Recent attacks exploit deeper microarchitectural state such as store buffers and line-fill-buffers [16, 62, 72, 73]. Checking software for the presence of these attacks requires analyses that take this microarchitectural state into account. However, owing to microarchitectural complexity, verification of software programs directly against the source RTL does not scale with the program being verified. Abstract formal models that only represent parts of the microarchitecture relevant to the attack while eliding the rest are simpler and more interpretable than the RTL. Verifying SW against such models instead of the source HW can improve scalability. However, accurate modelling is necessary for preserving strong guarantees; imprecise models can lead to software being deemed secure when in reality it is not.

Several recent works apply formal methods to verify software against hardware attacks [18, 29, 30, 55], and to identify new ones [23, 70]. These works perform verification against manually crafted abstract models that preserve only the vulnerability-relevant components of the HW design. However, manually writing models while ensuring the accuracy of preserved components is tedious and error-prone, and requires a deep understanding of the microarchitecture. While a security analyst can indicate which source design components they want to perform software-side security analysis against, it is hard to manually specify how these components should behave so that the model is sufficiently accurate. Thus, it is desirable to have a technique that, given some signals-of-interest, generates a formal model that accurately captures the behaviours of these signals w.r.t. the source RTL.

In this work, we propose micro-update models as a modelling formalism that can be used in security analysis of software running on the source RTL design. To address the challenges involved in hand-writing models, we develop a framework to mostly automatically generate micro-update models from RTL designs. The generated model achieves abstraction by capturing signals-of-interest specified by the user, eliding the rest, and achieves accuracy by ensuring functional equivalence of these signals w.r.t. the source RTL. This guarantees soundness, i.e. security properties over the signals-of-interest (e.g. information-flow/non-interference (NI) [21, 27]) that hold on the lifted model also holds on the RTL.

Imperative modelling of the functional behaviour of the signals-of-interest in the micro-update model: (a) aids correct-by-construction generation, since the model can be checked against the source RTL (e.g. with a functional equivalence proof) and (b) enables sound verification of software w.r.t. security properties (e.g. NI). In contrast, ISA-based models (e.g. [19, 28]), are equivalent only w.r.t. the architectural state, and lack the microarchitectural detail necessary for security analysis. On the other hand, approaches such as µspec [48, 49] formulate axiomatic ordering models that abstract away functional behaviours. Due to the lack of imperatively modelled state, µspec requires global invariants for checking accuracy with
the source RTL. Defining these invariants is challenging [35, 51]. Secondly, properties such as NI [21, 27], which requires a model that preserves functional behaviours cannot be expressed with these models. In conclusion, imperatively modelling functional behaviours of the signals-of-interest makes checking alignment with source RTL, and security properties on SW easier with the micro-update model.

Our micro-update model generation framework is based on lifting, which aims to extract high-level code from low-level source implementations while maintaining important properties with respect to the source. While lifting has been demonstrated in the PL and systems domains [3, 4, 41, 65], we extend it to hardware. We demonstrate how lifting can be performed hierarchically based on oracle-guided inductive synthesis [40, 63], thereby increasing its scalability. Our techniques aid in generating micro-update models from RTL, which then can be used for software security analysis.

**Contributions.** Our contributions are as follows:

- **Micro-update models as a new formalism:** We propose micro-update models as a formalism for abstract modelling of hardware designs. They provide finer temporal resolution than ISA-level models and preserve functional behaviour. Thus, they provide accurate abstractions of the hardware against which software can be verified for microarchitectural attacks.

- **Synthesis techniques for micro-update models:** We develop a methodology to generate micro-update models from RTL using a novel instantiation of oracle-guided synthesis [39, 40, 60, 63]. We show how one can generate models hierarchically, improving the scalability of synthesis, and provably maintaining a notion of equivalence with the RTL.

- **Empirical Demonstration:** We evaluate our approach on the Sodor5Stage processor [71] and components from the cva6 processor [1] by generating micro-update models for them. We demonstrate how applying the generated models to perform security analysis of software can lead to performance improvements over the source RTL.

**Outline.** In §2 we motivate and contrast the features of our modelling approach with existing ones through an example. In §3 we describe the structure of our models and their semantics. In §4 we describe the techniques that we use to generate our models. We discuss the experimental evaluation in §5. Finally, §6 discusses limitations, §7 is related work and §8 concludes.

## 2 Motivating Example

### 2.1 Example: modified Sodor5Stage Core

Sodor5Stage [71] is a 5-stage in-order processor implementing the RV32UI instruction set [75]. We augment this processor design with a new component, called a load buffer (LB) as shown in Fig. 1. The LB caches the most recently loaded value in its entry, which is valid when the address matches and the entry is not outdated. This cached value can be consumed by a subsequent load if its address matches and the entry is valid: canLoad @ (LoadAddr == addr) ∧ valid.

The LB entry is flushed by any subsequent store instruction to prevent reads from outdated entries. This feature does not change the ISA-level behaviour of the processor (program counter, register file and memory); its effects are purely microarchitectural. Now we discuss how a hypothetical attack can leak information by using the LB as a side channel.

### 2.2 The LB optimization vulnerability

Microarchitectural features such as the LB, can have several possible implementations. In Fig. 3(a,b) we consider two implementations for sw which differ in the exact cycle at which the LB is invalidated by the store. In Fig. 3(a) the invalidation happens during the exe stage of the sw instruction, while in (b) it happens during the mem stage. We call these implementations \(sw_{exe}\) and \(sw_{mem}\) respectively. An \(lw\) instruction uses a memory in the mem stage and refills the LB in the wb stage in both cases (Fig. 3(c)).

While architecturally invisible, such implementation differences crucially affect security analysis. To see this let us consider the effect of executing a load consecutively followed by a store, and compare them across the \(sw_{exe}\) and \(sw_{mem}\) implementations. In the case of \(lw; sw_{exe}\) (Fig. 2 top), the LB has a valid entry at the end of execution (refilled due to the lw instruction). On the other hand, in the case of \(lw; sw_{mem}\) (Fig. 2 bottom), the LB entry is flushed by the sw after the refill caused by lw. Consequentially, \(lw; sw_{exe}\) leads to an LB which is tainted (due to the lw), while \(lw; sw_{mem}\) does not.

---

1. Oracles are humans/tools that provide guidance to the synthesis engine in a specified format through a query-response interface.
A timing-based side-channel (e.g. Prime+Probe [25, 47]), can allow an attacker to infer the contents of the cache-like LB through a timing measurement. In such a case, the lw; sw instructed can lead to secret data (e.g. private keys) getting revealed to the attacker. On the other hand, since the lw; sw sequence leads to a sanitized LB, (i.e. the LB is untainted), a timing measurement cannot reveal victim secrets. Such microarchitectural details can render software insecure under certain implementations, while secure under others. Hence analyses checking software security must operate on models that expose microarchitectural detail.

2.3 Instruction-level approaches

Models that capture ISA-level behaviours [14, 19, 28, 66, 74] are precise only with respect to software-visible architectural state (e.g. program counter, register file, memory, and CSRs). This is also true of instruction-level-abstraction (ILA) based approaches [36, 38, 81] which model accelerators. However, hardware attacks manifesting at the microarchitectural (e.g., [16, 44, 46, 62, 72, 73]) level exploit software-invisible features (e.g. caches, predictors). Consequently, ISA-level models are restricted to certain software-level abstractions of security properties for these attacks. One example is the constant-time software contract, which requires instruction counts and memory access sequences to be independent of any secret information. While constant-time can be expressed over the architectural state, it is sensitive to the microarchitectural implementation [17]. Abstracting away microarchitectural detail can also lead constant time to impose overly strong requirements on software. In the example (Fig. 3), the instruction sequence lw; sw will be secure (assuming the lb_addr is the observable state) even if the lw loaded data from a secret-dependent address. This is because the sw would flush the lb_addr. However, since the load address is secret dependent, this code is not constant-time.

ISA-level approaches also subscribe to a temporal granularity that corresponds to the isolated execution of individual software (assembly) instructions. To illustrate this, let \([i_1; i_2]\) denote the effect (semantics) of the sequence of two instructions on the HW design. If one is only concerned with the architectural state, \(\sigma_{arch}\), the effect of \([i_1; i_2]\) is equivalent to the effect of these instructions executed in sequentially and in isolation, \([i_1]\) followed by \([i_2]\) (i.e., \(i_1\) finishes execution before \(i_2\) begins):

\[
\sigma_{arch} [i_1; i_2] \sigma'_{arch} \iff \sigma_{arch} [i_1] \sigma'_{arch} [i_2] \sigma'_{arch}
\]

However, the effect of a stream of instructions with respect to the microarchitectural state \(\sigma_{arch}\) can be different than that of individual instructions operating in isolation:

\[
\sigma_{arch} [i_1; i_2] \sigma'_{arch} \iff \sigma_{arch} [i_1] \sigma'_{arch} [i_2] \sigma'_{arch}
\]

In the LB example, the semantics of both \(sw_{exe}\) and \(sw_{mem}\) are identical when executed in isolation. This is true even with respect to the LB state, as both flush the LB before finishing execution. However, the semantics of \(sw_{exe}; lw\) and \(sw_{mem}; lw\) are different owing to differences in microarchitectural implementation.

In summary, models that define semantics for isolated execution between instruction fetch and commit points (orange lines in Fig. 3) fall short when representing the microarchitecture. Thus, we need a model that accurately captures the microarchitectural interactions between instructions.

2.4 Axiomatic ordering-based approaches

Approaches such as µspec [48] model the microarchitecture as a set of axiomatic ordering constraints over the execution of a program. Executions satisfying these constraints are valid (can be observed) while others cannot. Since these constraints are over full executions, µspec can express behaviours of instructions executing simultaneously (as in a pipeline), and does not suffer from the isolation problem discussed earlier.

Ordering-based approaches often abstract away functional behaviour. However, functional behaviour is necessary to model low-level security-relevant microarchitecture, as we now discuss. Consider a victim program running on a processor with an LRU-cache. Such processors can leak victim secrets through the LRU state [79]. Detecting such vulnerabilities in the victim program requires security properties that precisely identify memory access sequences which lead to attacker-distinguishable LRU states. A non-interference (NI) [27, 69] property identifies such accesses by requiring that subsequent attacker-observable cache hit/miss outcomes should not depend on victim secrets. However, since expressing security properties such as NI requires a model that exposes the functional behaviour of signals, it is infeasible with ordering-only models. A micro-update model can represent the low-level LRU policy functionally, enabling NI-based analysis.

µspec-based security analysis approaches (e.g. [55, 70]) work around this by explicitly identifying execution fragments which are indicative of vulnerabilities, e.g., exploit patterns ([S3A-II in [70]) or transmitters ([S3.2.1 in [55])]. While NI uniformly captures all possible vulnerable cases as a single property, these execution fragments (patterns/transmitters) must be manually enumerated. This requires a deep understanding of the microarchitecture and can result in false negatives if some vulnerable execution fragments are missed.

The second advantage of the micro-update model is that it is operational. That is, it represents executions as imperative transitions performed on some state at every step. This allows checking per-step equivalence (§4.2) with the RTL design using a model checker (e.g. SymbYosys [20], JasperGold). Such checks are essential to ensure accuracy when lifting a model. Ordering-based models, on the other hand, are axiomatic and define behaviours in terms of declarative constraints. Checking these constraints against the RTL requires global invariants (i.e. invariants over full executions as opposed to individual steps). These invariants can be difficult to generate and check automatically (e.g. [35, 51]), which in turn makes checking accuracy with the source RTL more difficult.

In conclusion, the micro-update model (a) represents executions at a finer granularity than ISA models, (b) preserves functional behaviour enabling NI-based security analysis and (c) represents executions operationally, thus allows easier equivalence checking with the source RTL.

3 Micro-update Models

In this section, we illustrate the micro-update model through Figure 4. A micro-update model is defined over a set of signals, and has two kinds of elements: micro-updates and guards. A micro-update is an imperative code snippet that defines a functional operation performed on signals from the model. For example, the flush micro-update (lw, fsh) in Fig. 4(B) operates on the LB (from Fig. 1): it clears the data and address fields and sets the valid bit to zero.
Guards are Boolean predicates that control when a micro-update is triggered. Guards are defined over a set of trigger signals. For example in Fig. 4(C), the guard for lb_flush is true and lb_flush is executed at cycles where the instruction in the memory stage (instr_queue[2]) is a store. Trigger signals typically are top-level inputs to the source design. Signals from the model map to signals in the source RTL design. Thus, the model as a whole captures a slice of the design, as identified by these signals-of-interest (§3.1.1).

**Example 3.1.** Consider an addi instruction executing on the simple pipelined processor in Fig. 4(A). During execution addi interacts with several pipeline components (e.g. decode unit), by invoking multiple micro-updates. Each micro-update is executed when the corresponding guard (Fig. 4(C)) evaluates to true (e.g. decode_itype is executed when instr_queue[2]), the decoded instruction, is an I-type instruction). While instr_queue[2] depends on a single instruction input, in general, guards can depend on multiple inputs. For example, if rs1_addr (instr_queue[1]) == rd_addr (instr_queue[2]), then there is a data dependency between the current and previous instructions. In this case, the alu_compute_alu_out_bypass_imm micro-update is invoked (ref. Fig. 4(D)) instead of alu_compute_rs_imm (which uses data from the RF). While we take the example of addi, the model can be extended to cover other instructions. The resulting model will capture a thicker slice of the RTL with more signals, micro-updates, and guards. We illustrate example micro-updates for sw in Fig. 4(D).

We now discuss how this model addresses concerns from §2.

**Finer temporal resolution.** ISA-based approaches define semantics based on instruction-level temporal resolution, i.e., between fetch and commit points of an instruction. The micro-update model decomposes the execution of each instruction into several micro-updates, and evaluates guards and micro-updates on a per-cycle basis. Consequently, a micro-update model has a finer temporal resolution than ISA-based models by identifying the exact cycle at which a microarchitectural state update happens (blue markings in Fig. 3). This allows the micro-update model to discriminate between different microarchitectural implementations that are architecturally indistinguishable. For example, it can differentiate between the $sw_{exec}$ and $sw_{mem}$ implementations (as introduced in §2.2), and specifically identify ($lw, sw_{exec}$) as being vulnerable.

**Imperatively modelled functional detail.** While axiomatic modelling approaches (such as jspec) can also provide finer temporal resolution (through microarchitectural events), they often lack functional detail. A micro-update model, however, captures functional behaviour through micro-update bodies (Fig. 4(B)). As discussed in §2.4, this allows specification of security properties like NI. Secondly, micro-updates describe imperative operations on the model state (§3.1.1). Imperative modelling enables per-cycle equivalence proofs with the RTL, which is essential for accurate model lifting. This contrasts with axiomatic approaches which require global invariants ([35, 51]).

### 3.1 Components of the micro-update model

#### 3.1.1 Signals

A micro-update model captures a design slice that is identified by a set of signals-of-interest, which we denote as $S_{data}$. When generating the model, the user can specify these based on the RTL signals over which they wish to perform security analysis. At each cycle, each data signal from $S_{data}$ is mapped to a value by an assignment $\sigma_{data} : S_{data} \rightarrow V$.

#### 3.1.2 Micro-updates

A micro-update is an imperative code block that typically operates on small subsets of signals (Fig. 4(B)). The model as a whole comprises a set of micro-updates, $L$. At each cycle, signals from $S_{data}$ are updated based on the micro-updates from $L$ that are invoked during that cycle. The body of each micro-update $t \in L$ provides the semantics $\ll t \rr$, which defines how $t$ transforms the $S_{data}$ signals when invoked. An instruction typically performs several of these micro-updates during its lifetime, as Fig. 4(D) illustrates.

#### 3.1.3 Trigger signals and guards

The micro-updates are invoked based on a Boolean condition called a guard. A guard for a micro-update $t$ is a boolean condition evaluated over the trigger signals $S_{trig}$. We view this as a function $G_t(S_{trig}) \in \{true, false\}$. The micro-update $t$ is executed at a cycle if and only if $G_t(S_{trig})$ evaluates to true at that cycle. Trigger signals typically correspond to top-level inputs of the RTL design (e.g. instruction inputs, memory ports). Thus the micro-update...
model can directly relate micro-update invocation to the instructions provided to the design. This is beneficial since it allows micro-update models to be easily connected to software-side analyses (which operate over ISA instructions).

3.1.4 Execution of the micro-update model
As introduced in §3.1.1, states of the micro-update model transition system are assignments to the signals-of-interest, \( \sigma : S_{\text{data}} \rightarrow \mathcal{V} \). We denote \( \Sigma_{\text{data}} = S_{\text{data}} \rightarrow \mathcal{V} \) as the set of all assignments. For initial state \( \sigma_{\text{init}} \) and a sequence of inputs \( \mathcal{I} \) to the model, an execution is the sequence of states that the model is in at each cycle. We denote this as \( \{ [\mathcal{I}]_{m}(\sigma_{\text{init}}) \in \Sigma_{\text{data}}^{*} \}. \) Here \( m \) indicates that this is an execution (or trace) of the model. Later we will use \( d \) to indicate executions of the source RTL design.

3.2 Properties of micro-updates

Uses and modifies Knowledge of the signals used and modified by each micro-update allows us to optimize the synthesis of the micro-update model (§4.4.2). This information can be extracted from the micro-update code blocks (for example, by an AST traversal). We denote the set of signals used and modified by micro-update \( t \) as \( \text{use}(t) \) and \( \text{mod}(t) \) respectively. For example, the \texttt{lb_refill} micro-update from Fig. 4(b) has \( \text{use}(t) = \{ \text{mem_addr, mem_data} \} \) and \( \text{mod}(t) = \{ \text{lb_addr, lb_data, lb_valid} \} \).

Sequential and combinational micro-updates We allow micro-updates of two types: combinatorial and sequential. These resemble sequential and combinational logic seen in Verilog-like HDLs and have similar semantics. Sequential updates consume values from the previous state, while combinational updates use values from the current cycle. Combinational updates are evaluated after dependencies have stabilized (i.e. like RTL). Presence of combinational logic in the RTL (e.g. adders) requires combinational modelling to catch the correct values.

\( \mathcal{M}-\text{set} \) While each micro-update only changes a few signals, more than one micro-update can be triggered in a single cycle of execution. We use the term \( \mathcal{M}-\text{set} \) for such a set of micro-updates that can be triggered at the same cycle of execution. Not all sets of micro-updates are valid. For an \( \mathcal{M}-\text{set} \) to be valid, there should not be any combinational cycles, and the constituent micro-updates must modify disjoint signals:

\[
\forall t_1, t_2 \in M. \mod(t_1) \cap \mod(t_2) = \emptyset
\]

Both these conditions can be checked from the uses and modifies information. The effect of triggering an \( \mathcal{M}-\text{set} \) follows the usual sequential and combinational semantics.

4 Micro-update Model Synthesis

In this section, we discuss our framework (Fig. 5) to generate micro-update models using formal synthesis techniques. We begin by discussing the key challenges that we face.

4.1 The challenge posed by formal synthesis
While (semi-)automated approaches to synthesize formal models from RTL are useful, formal synthesis is challenging (more so than formal verification), since it requires a search over model candidates, while also verifying them for correctness. Synthesizing models involving ordering/timing constraints (e.g. for hardware) is especially challenging (compared to program synthesis [53, 67]) since the generated model must capture temporal constraints in addition to functional correctness. We navigate this complexity by decomposing the synthesis objective into a monolithic model into smaller functional (micro-updates) and temporal (guards) components.

Since individual micro-updates operate on small, local subsets of signals, identifying them is easier compared to identifying the right coordination between micro-update invocations. The latter requires a deep understanding of inter-dependencies between micro-updates. This is hard to do manually; e.g. ~10 out of 42 RTL-bug issues in the cva6 processor [2] were due to imprecisely coordinated inter-dependencies. Our synthesis procedure does the heavy lifting of determining this choreography of micro-update invocations.

4.2 Problem formulation and guarantee
Our synthesis framework allows the user to specify the signals \( S_{\text{data}} \) which they wish to be captured by the generated model. The choice of \( S_{\text{data}} \) exposes the tradeoff inherent to the development of formal models: models for a larger \( S_{\text{data}} \), while more detailed, are harder to synthesize and analyze, and vice-versa. Given an RTL design, the signals \( S_{\text{data}} \) and library \( L \), our goal is to generate guards \( G_t \) for each \( t \in L \), so that the behaviours of identified signals-of-interest \( (S_{\text{data}}) \) are equivalent to behaviours of corresponding signals in the source design (Property 1).

\( S_{\text{data}} \)-equivalence We now formalize the above property, which we call \( S_{\text{data}} \)-equivalence. Recall that we defined states of the micro-update model as assignments to \( S_{\text{data}} \) (§3.1.1) and executions as sequences of these assignments (§3.1.4). Similarly, we also view the source design as a transition system over all the signals from the design, \( S \). These are a superset of signals captured in the model, \( S \subseteq S_{\text{data}} \). States in the transition system of the source design are assignments to signals in \( S, \sigma : S \rightarrow \mathcal{V} \). For a design state \( \sigma_d \) and a model state \( \sigma_m \), we denote that \( \sigma_d \) and \( \sigma_m \) agree on all values in \( S_{\text{data}} \) as \( \sigma_d \cong S_{\text{data}} \sigma_m : \)

\[
\sigma_d \cong S_{\text{data}} \sigma_m \Delta \forall s \in S_{\text{data}}. \sigma_d(s) = \sigma_m(s)
\]

Finally, similarly to the micro-update model, we denote a trace of the design starting from state \( \sigma_{\text{init}} \), on input \( \mathcal{I} \), as \( \{ [\mathcal{I}]_{d}(\sigma_{\text{init}}) \}. \) Our property states that when execution begins from initial states of the design, \( S \) are the transition system of the source design are assignments to signals in \( S, \sigma : S \rightarrow \mathcal{V} \). For a design state \( \sigma_d \) and a model state \( \sigma_m \), we denote that \( \sigma_d \) and \( \sigma_m \) agree on all values in \( S_{\text{data}} \) as \( \sigma_d \equiv S_{\text{data}} \sigma_m : \)

\[
\sigma_d \equiv S_{\text{data}} \sigma_m \Delta \forall s \in S_{\text{data}}. \sigma_d(s) = \sigma_m(s)
\]

This property is guaranteed to hold because of the equivalence check performed as the final signoff on the model (§4.6).

Hyperproperty-preservation under \( S_{\text{data}} \)-equivalence The \( S_{\text{data}} \)-equivalence property allows us to infer, in a sound manner, that a program securely executes on the design if it securely executes on the model. We formalize this reasoning by recalling non-interference based security hyperproperties.

Let \( S_{\text{pub}}, S_{\text{obs}} \) be subsets signals representing public inputs and (attacker)-observable outputs. A non-interference property says
We refer the reader to literature (e.g., [21]) for more details. While we eliminate these, we move to the second phase: guard synthesis. We generate simulation traces (§4.4.1) and use these traces to generate candidate M-sets (§4.4.2). These M-sets, while consistent with the simulation traces, may contain spurious candidates that do not apply generally. We use two techniques to weed out spurious candidates: distinguishing oracles and cover properties (§4.4.3). Once we eliminate these, we move to the second phase: guard synthesis (§4.5). Guard synthesis uses the M-sets from the previous phase as well as the set of trigger signals as examples when synthesizing guards for each micro-update. In this sense, guard synthesis is a form of example-based/inductive synthesis [33, 63]. Guard synthesis results in a complete candidate micro-update model. This candidate model is checked for \( S_{data} \)-equivalence (§4.6).

4.3.1 Signals-of-interest and RTL-mapping

The synthesis procedure takes as input the signals \( S_{data} \) (§3.1) to be included in the micro-update model, and a mapping from these signals to corresponding signals in the RTL design. The generated model then satisfies the \( S_{data} \)-equivalence property (Property 1).

4.3.2 Micro-update library

We also require the user to provide a library of micro-updates (L in Fig. 5). Candidate M-set generation (§4.4) searches over L to construct valid M-sets for the simulated traces. Since micro-updates operate on small subsets of signals, specifying this library often only requires a local understanding of the design. In our experiments, we observed that library specification time was negligible compared to design harnessing/instrumentation.

4.3.3 Discussion on the choice of micro-update library

Specificity and canonicity of micro-update library The micro-update library used when generating a model for a set of signals \( S_{data} \) should be able to capture all possible single-cycle transitions of the signals in \( S_{data} \). As such, these transitions depend on the design. For example, consider two processor designs, of which one implements bypassing while the other does not. The latter would need additional micro-updates that account for bypassing, by allowing the ALU output to be a function not only of register file outputs but also pipeline registers for stages further along in the pipeline. However, the micro-update library need not be specific to a single design. Multiple designs from the same design family can share micro-updates between them. In our experiments, we observe that the Sodor family of processors share micro-updates corresponding to instruction decode, the register file, and functional units. However, more pipelined processors require additional micro-updates to account for more intermediate stages. Finally, we note that a larger-than-necessary micro-update library does not affect the correctness of the generated model. However, as the search space increases with a larger library, it may affect the synthesis runtime. We now discuss this further.

Sensitivity of micro-update library to synthesis We now comment on the sensitivity of synthesis to the user-provided micro-update library. Firstly, libraries which have incorrect or irrelevant micro-updates (e.g., LB micro-updates for a model with only ALU operations) does not affect the soundness of the generated model. Such micro-updates are filtered out by candidate M-set generation, i.e., their guard is assigned false. Since this phase does not invoke (costly) solvers, synthesis performance is also not heavily affected.
Fig. 6. A sample execution trace from simulation.

Libraries which are incomplete, i.e. do not contain all micro-updates necessary to justify the simulation traces, fail during M-set generation. In such cases, our approach identifies the failing simulation step and signals. This helps to add missing micro-updates.

4.4 Generating M-sets

4.4.1 Simulator-based trace generation

We generate a set of random test traces of the RTL execution from a simulator (e.g. iverilog [77]). These traces must include all the signals from $S_{data}$. Figure 6 shows a fragment of such a trace which includes signals for the memory port and the LB component described in §2.

4.4.2 Generating candidate M-sets

A candidate M-set is a set of micro-updates that transforms the assignment to $S_{data}$ some cycle $i$ of a trace into the assignment at the next cycle $i + 1$. As the first part of the synthesis procedure, we extract candidate M-sets that match the simulation traces. For each step of the simulation traces, we obtain a pre-post assignment pair, $(\sigma_{data}, \sigma'_{data})$. For example, from the second step in Fig. 6 we extract: $\sigma_{data} = [lb_{data} \mapsto 10, \cdots]$ and $\sigma'_{data} = [lb_{data} \mapsto 30, \cdots]$. The generation of candidate M-sets for $(\sigma_{data}, \sigma'_{data})$ is performed via a depth-first-search (DFS) over micro-updates from the library. The DFS explores a sequence of micro-updates, which incrementally transforms $\sigma_{data}$ into $\sigma'_{data}$. A naive DFS will consider all possible sequences of micro-updates. However, typically, several micro-updates do not depend on each other. We exploit this notion of independence to eliminate redundant search.

In general, we say that micro-update $t'$ depends on micro-update $t$ if $t'$ is combinational and use($t'$) $\cap$ mod($t$) is non-empty. Since sequential micro-updates only consume values from the previous step, they are not dependent on anything. Subsets of independent micro-updates can be searched independently in the DFS, which avoids redundant orderings.

Search pruning is also performed based on the fact that only one micro-update modifies a given signal at each step. Hence, after choosing a micro-update modifying a signal $s$, the DFS ignores all other micro-updates modifying $s$. While the number of micro-update subsets is exponential, these strategies avoid search-space explosion, making candidate M-set generation robust to the size of the micro-update library $L$.

Even though each candidate M-set generated by the DFS transforms the assignment $\sigma_{data}$ into $\sigma'_{data}$, some of these candidates may be spurious, as we now illustrate.

Example 4.2 (Spurious M-sets). Consider the trace in Fig. 6. For the third transition (counter = 2 to 3), the lb_refill micro-update (Fig. 4B) correctly updates the LB signals. However, the lb_hold micro-update (not shown) which maintains the same values would also work for this cycle since the signals do not change. The DFS procedure will generate both of these as candidates. However, we note that in the design (Fig. 3) the LB is only refilled when the mem stage instruction, inst_mem, is a load. This is not the case for this transition since inst_mem is an addi instruction. Hence, lb_refill is a spurious candidate which we need to filter out.

4.4.3 Eliminating spurious M-sets

We eliminate such spurious M-sets using (a) distinguishing oracles (§4.4.3) and (b) cover properties (§4.4.3).

Using a distinguishing oracle

A distinguishing oracle [39] identifies a test program (and its corresponding trace) on which two given M-sets, $M_1, M_2$, differ in their behaviour. Running M-set generation on distinguishing traces only generates one of the two M-set candidates, eliminating ambiguities such as the one in Example 4.2.

Example 4.3 (Ex. 4.2 continued). If applied to the M-sets $M_1$ and $M_2$ which include micro-updates lb_hold and lb_refill respectively, a distinguishing oracle generates a trace that includes the first transition (counter=0 to 1) of Example 4.2. We note that only the lb_hold micro-update applies for this transition (since the LB entry remains invalid). The distinguishing oracle can also generate a trace with the second transition (counter=1 to 2), where only lb_refill applies. The distinguishing oracle can be used to specifically identify traces where $M_1$ (or $M_2$) is applicable and the other is not.

Adapting the technique from [39], a distinguishing oracle can be implemented by invoking a hardware model checker (e.g. SymbiYosys [20]) as follows. Given candidate M-sets $M_1, M_2$, we create two copies of the micro-update model. The first copy invokes micro-updates from $M_1$ while the second from $M_2$. Let $Uses$ be signals that are used by some micro-update in $M_1 \cup M_2$ and Mods be signals that are modified by both $M_1$ and $M_2$. Then, we invoke a hardware model checker to generate a trace such that at some cycle $i$, the two copies have equal values of all used signals,

\[
\text{assume}(\forall s \in Uses(\tau[i](s_1) = \tau[i](s_2)))
\]

while at the next cycle (i + 1), $M_1$ and $M_2$ generate different values for at least one modified signal,

\[
\text{assert}(\forall s \in Mods(\tau[i+1](s_1') \neq \tau[i+1](s_2'))) .
\]

Since $M_1$ and $M_2$ generate different values for some signal, only one of these can be a valid candidate M-set for the transition from cycle $i$ to $i + 1$. Thus the trace generated by the model checker distinguishes $M_1$ and $M_2$.

A typical case (§5.2) where the distinguisher helps is when a functional unit operates on one of several possible inputs (e.g. in the case of bypassing). In such cases, one can identify traces in which a specific bypass path is invoked.

Using cover properties

For a Boolean formula $\phi$, cover properties of the form $\text{cover}(\phi)$ can be supplied to a hardware verification tool (e.g. JasperGold, SymbiYosys [20]). The tool, for a given cover property $\phi$, aims to generate an execution in which $\phi$ holds. While the distinguishing oracle is useful when identifying traces on which specific M-sets do/do not apply, cover properties can be used to identify traces where certain guard predicates do/do not evaluate
to true. Traces satisfying the cover property can then be added to the trace corpus for future $M$-set generation. This allows the generation of rare executions that were not seen during the initial simulation.

**Example 4.4** (Using cover properties). Consider a simple 3-stage pipeline where the trigger signals correspond to the three instructions in the pipeline: $S_{\text{trig}} = \{i_{\text{fei}}, i_{\text{exc}}, l_{\text{sh}}\}$. Consider a bypassing path micro-update that is triggered when there is a data dependency between $l_{\text{exc}}$ and $i_{\text{fei}}$, and $l_{\text{exc}}$ does not write to zero. The guard corresponding to this micro-update is:

$$\phi_{\text{bypass}} \equiv (rd(l_{\text{exc}}) = rs1(f_{\text{fei}})) \& (rd(l_{\text{exc}}) \neq 5'b00000)$$

Since this condition requires matching register source/destination fields, the probability of it holding is somewhat low ($\sim 1/32$), and it may be missed during random simulation ($\S 4.4.1$). In such cases, the lack of example transitions for the bypassing path would result in the micro-update not being represented in the generated model. The cover property $cover(\phi_{\text{bypass}})$ generates an execution in which this path is triggered, mitigating this. Thus, the user can generate rare executions not observed in simulation.

### 4.5 Guard synthesis

The $M$-sets generated in the previous stage indicate micro-update invocations that specifically produce simulation traces. Guards ($\S 4.1.3$) generalize this to all traces. Intuitively, the $M$-sets generated in the previous stage serve as examples for guard synthesis. Guard synthesis generates guard expressions that are consistent with these examples. In this sense, guard synthesis works like programming by examples [$32, 64$]. Guard synthesis produces a candidate micro-update model which is then checked ($\S 4.6$) to satisfy Property 1.

We formulate guard synthesis using Syntax Guided Synthesis (SyGuS) which is a type of formal synthesis. The formal synthesis problem aims to generate an implementation of an object (typically a function) such that the generated implementation satisfies some given specification. SyGuS [$6$] builds on this core idea by restricting the search space of synthesis function implementations to terms from a context-free grammar and where the specification for the synthesis function is provided in the SMT-LIB format [$10–12$]. SyGuS is supported by multiple synthesis solvers (we use CVC5 [$9$]).

We formulate the guard $G_t$ for micro-update $t$ as a synthesis function over the trigger signals. For guard $G_t$ we extract pairs $(\sigma_{\text{trig}}^i, b^i)$ from the simulation traces and corresponding $M$-sets. Here, $\sigma_{\text{trig}}^i$ is the trigger signal assignment at step $i$, and $b^i$ is a Boolean representing whether $t$ belongs to a valid $M$-set at step $i$. We ensure that the synthesized guard is consistent with these pairs through the following synthesis constraints.

The first constraint is that for step $i$, if $b_i$ is false (meaning that micro-update $t$ wasn’t a part of any valid $M$-set), then the guard $G_t$ must evaluate to false on $\sigma_{\text{trig}}^i$: $\exists_i (b^i \lor \neg G_t(\sigma_{\text{trig}}^i))$. If not, the generated model would be incorrect at step $i$.

Additionally, for each signal $s \in S_{\text{data}}$, we require that exactly one micro-update modify $s$ at each step. We define the set of micro-updates that modify $s$ as $\text{mods}(s)$. The following formula enforces the “exactly one” constraint for signal $s$:

$$\bigwedge_{t \& t \equiv \text{mods}(s)} (\neg G_t(\sigma_{\text{trig}}) \land \neg G_t(\sigma_{\text{trig}})) \land \bigvee_{t \equiv \text{mods}(s)} G_t(\sigma_{\text{trig}}).$$

On posing this problem to a SyGuS solver, the solver returns expressions for each guard $G_t$ in terms of the trigger signals. This guard expression, together with the micro-update bodies, gives us a complete model. The inability of the solver to synthesize some guard implies that the set of predicates was insufficient, i.e., there are factors outside this set that the guard depends on. The user can then try strategies such as adding more trigger signals, or increasing the SyGuS grammar depth.

### 4.6 Equivalence checks

Finally, the generated model is checked against the source design for $S_{\text{data}}$-equivalence (Property 1). If the equivalence proof fails, we get back a counterexample trace. This trace is fed back into the trace database and the synthesis loop is repeated. Future iterations of synthesis use this new trace, and hence avoid synthesizing the same (incorrect) model.

### 4.7 Hierarchical synthesis

The approach discussed so far monolithically generates micro-update models from the source RTL. However, our approach can benefit from the hierarchy inherent to RTL designs, which is what we now discuss. Consider a micro-update model with signals $S_1$ generated from a design component $C_1$. Now suppose we want to generate a model with signals $S_2$ for a (larger) component $C_2$, of which $C_1$ is a sub-component. By reusing the model for $C_1$, we only need to generate the slice of the model corresponding to signals $S_1 \setminus S_2$ (i.e., signals in $S_1$ but not $S_2$). In particular, this requires generating guard predicates only for micro-updates relevant to $S_1 \setminus S_2$. This decomposition leads to smaller synthesis queries, thus improving performance.

Hierarchical synthesis, however, requires that (a) the signals $S_1$ from the sub-component ($C_1$) be not directly driven by logic outside the component, and (b) that the trigger signals for $C_1$ map to trigger signals for the parent component $C_2$ in a straightforward way. Here (a) ensures that $S_1$ signals preserve their behaviour in the larger model, and (b) ensures that the guards from $C_1$ in terms of trigger signals from $C_2$. We demonstrate an application of this in $\S 5.3.2$, where we first generate a model for the $\text{store}\_\text{unit}$ of cva6 ($C_1$). Then we reuse it when generating a model for the $\text{load}\_\text{store}\_\text{unit}$ ($C_2$), of which the $\text{store}\_\text{unit}$ is a sub-component.

### 5 Experimental Evaluation

#### 5.1 Methodology

We implement our framework in a Python-based tool called PAUL (Python-based Atomic-Update Language toolkit). The user specifies signals-of-interest ($\S 4.3.1$), a library of micro-updates over these signals ($\S 4.3.2$), and a set of predicates for the guards. The tool allows the user to simulate the design over input programs, generate $M$-sets ($\S 4.4$), and synthesize guards ($\S 4.5$).

**Hyperparameters** While Fig. 5 indicates the general synthesis pipeline, in practice, there are knobs that can be used to guide the synthesis. Both the distinguishing ($\S 4.4.3$) and cover ($\S 4.4.3$) trace generation require calls to a model checker, which can be time-consuming. The user can selectively apply these checks. For the distinguishing oracle, this amounts to choosing two $M$-sets to distinguish between, while for the cover property oracle, this amounts to providing a (possibly partial) predicate valuation.
**Backend setup**  PAUL interfaces with iverilog [77] for simulation, and the Yosys [78] based SymbiYosys [20] for model checking. SymbiYosys uses Boolelor [56] and ABC [5] as backend solvers. We perform experimentation on an Intel Core i7-10610U processor at 2.3GHz with 16GB RAM.

**Evaluation overview**  We conduct two case studies by synthesizing micro-update models for: (a) Sodor5stage (§5.2) and (b) components from cva6 (§5.3.1). We demonstrate how the hierarchical lifting of micro-update models (§5.3.2) can improve scalability. We showcase applications of the lifted models for security verification of software. We demonstrate better performance compared to verification against the source RTL, and strong soundness guarantees (which are often lacking in hand-written models).

### 5.2 Case Study: The Sodor5stage processor

The Sodor5stage is a 5-stage in-order processor [71] with a simple scratchpad memory. We augment this design with the load buffer (LB) feature (as discussed in §2). We now discuss the results of model lifting, referring to Table 1.

We generate three models from the processor corresponding to different ISA subsets (rows in Tab. 1): ALUI (ALU immediate), ALUR (ALU register), and ALULS (ALU immediate + memory instructions). In each case, we identify a set of signals (S_data) and the library of micro-operations (L). In all models, we have a queue of the previous five instructions as the control state. Since this is a 5-stage design, this suffices to represent all inter-instruction interactions.

**M-set generation**  In each case, we perform simulation over the respective instruction subsets by constraining the opcode. We extract the transitions and generate the candidate micro-update M-sets for these simulation traces. We observe that for the comparison instructions (e.g. s1t), comparisons of several bypassing-path signals have the same (0/1) result leading to spurious M-sets. We provide the counts of distinguishing and cover traces used to eliminate these spurious cases in Table 1. As M-sequence generation does not involve (costly) solvers, it requires much less time (<5s) as compared to guard synthesis.

**Guard synthesis and equivalence**  In the second phase of the synthesis, we use the M-sets to synthesize guards by formulating a SyGuS [6] query. We use cvc5 [9] as the SyGuS solver. Finally, we check the candidate micro-update model for S_data-equivalence with respect to the source RTL. We perform this check using a bounded model checking (BMC) query with SymbiYosys ([20]) model checker. We use a depth of 15 steps for BMC which ensures that all possible inter-instruction interactions across the pipeline are explored.

5.2.1 Do lifted models aid software security analysis?

In this section, we explore whether/how lifting micro-update models improves the trustworthiness and performance of security analysis. We begin by recalling the Bounds-Check-Bypass (BCB) gadget from the Spectre vulnerability [43, 44].

**Spectre BCB background**  Figure 7(A) shows the Spectre BCB gadget and Fig. 7(B) shows assembly fragments for the array2 access in the gadget. Fig. 7(C) shows the assembly fragment after a minor modification to the gadget (changing & = to & =). In B (but not C), two ALU instructions separate the lw instruction (101b) from the sw instruction (101c). Inlays in Fig. 7 depict timing diagrams for executions of B,C on Sodor5stage with the LB modification, under the swexe implementation. In B, swexe flushes the LB after the lw refills it. However, in C, swexe flushes LB before the refill. If the 1b_addr were attacker observable (as discussed in §2.2), B would be safe, while C would not.

This example shows how small changes to the microarchitecture such as the LB can render the hand-written abstract models adopted by existing approaches [18, 23, 29, 30] imprecise. As an example, the abstract model from [18] which is designed to check Spectre variants would flag both Fig. 7(B), (C) as vulnerabilities, which are not. In B, (C) are spurious cases in Table 1. As M-sequence generation does not involve (costly) solvers, it requires much less time (<5s) as compared to guard synthesis.

**Semantic information-flow experiment**  We now demonstrate how the lifted (ALULS) model can be used to perform semantic information-flow analysis on software. We check whether there exists an information-flow from certain source (src) signals to the 1b_addr signal when executing small litmus test programs. Note that 1b_addr is assumed to be adversary-observable (§2.2). If this check passes, it guarantees that victim secrets (e.g. private keys) are not leaked through the LB side-channel when the test program is executed. We formulate information-flow as a variant of the non-interference [21] hyperproperty. We check this by invoking the SymbiYosys model checker.

Table 2 draws a comparison of the run times of these information-flow checks when performed against the lifted model and the source design. Each check (row in Tab. 2) is performed on a 1-4 instruction symbolic litmus tests (where the opcode is fixed and other fields are unconstrained). We observe that verification against the lifted model results in improved performance over the source RTL in most cases. We can guarantee the soundness of this analysis since the lifted model preserves S_data-equivalence with the source model (Property 1). This demonstrates that model lifting can improve the trustworthiness of security analysis over hand-written models while providing performance improvements over the source design.

### 5.3 Case Study: The cva6 (Ariane) processor

In this section, we apply our lifting methodology to components from the cva6 (Ariane) processor [1]. Ariane is a 6-stage application-class processor implementing the RISC-V 64-bit instruction set.

```c
void victim_function_v01 (unsigned int x) {
  if (x < array1_size) temp = array2[array1[x] * 512];
}
```

Figure 7. Subtle program changes can expose vulnerabilities: (A) Ex. 1 from Kocher’s [43] examples that is vulnerable to a Spectre [44] attack. (B, C) Compiled assembly from the original example, and the example after changing the & = to & =. The accompanying waveforms show how (C) remains tainted under swexe while (B) does not.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUI</th>
<th>ALUR</th>
<th>ALULS</th>
</tr>
</thead>
<tbody>
<tr>
<td>101b:</td>
<td>lw</td>
<td>a4, 0(a5)</td>
<td></td>
</tr>
<tr>
<td>101b:</td>
<td>lw</td>
<td>a5, -100(gp)</td>
<td></td>
</tr>
<tr>
<td>101b:</td>
<td>and</td>
<td>a5, a5, a4</td>
<td></td>
</tr>
<tr>
<td>101b:</td>
<td>sext. w</td>
<td>a4, a5</td>
<td></td>
</tr>
<tr>
<td>101c4:</td>
<td>sw</td>
<td>a4, -100(gp)</td>
<td></td>
</tr>
<tr>
<td>101b6:</td>
<td>add</td>
<td>a5, a5, a4</td>
<td></td>
</tr>
<tr>
<td>101b6:</td>
<td>lw</td>
<td>a4, 0(a5)</td>
<td></td>
</tr>
<tr>
<td>101b6:</td>
<td>sw</td>
<td>a4, -100(gp)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Comparison of run times of information-flow checks against the lifted model and the source design.
We now briefly describe the components we lift (for details see \cite{1, 54}). The TLB \cite{8, 59} is housed inside the memory management unit (MMU) of the processor and uses a PLRU (pseudo-least-recently-used) eviction scheme \cite{58}. The write-buffer (wbuffer) is a coalescing buffer for the write-through data-cache, and sits between the core, data-cache and data-memory. The store unit (store_unit) maintains a queue of pending store requests from the core. Requests are initially queued into a speculative queue and are later moved to a commit queue (upon receiving a commit signal from the core). The load-store unit (load_store_unit) consists of the load and store unit sub-modules. It only handles one request at a time and stalls if there is a load-after-store conflict with an unfulfilled store operation (at the same address).

5.3.1 Generating micro-update models for cva6

We now discuss the generation of micro-update models for the aforementioned components, summarizing results in Tab. 3.

**TLB and wbuffer** For the TLB and wbuffer, we generate micro-update models in which the signals-of-interest ($S_{data}$) include all entries in these buffers. For example, each wbuffer entry has three 1-bit signals representing its state: $s = \{valid, dirty, txnb1k\}$. The micro-update library L consists of five micro-updates that: (1, 2) receive a fresh write request or a replayed request to the same address, (3, 4) initiate or conclude a memory transaction, and (5) a nop (no operation). As reported in Tab. 3 while guard synthesis takes the maximum time of all the steps, the overall generation time of either model is less than ~4m.

**store_unit and load_store_unit** The load-store unit handles inter-instruction dependencies while interfacing between the core (for requests/commits) and the memory (for requests/responses). It is much more complex than the TLB and wbuffer components. This is reflected in the micro-update model synthesis times (Tab. 3). While the store_unit can be synthesized monolithically, monolithic guard synthesis for the load_store_unit hits a time out († in Tab. 3) of 1 hr.

5.3.2 Can hierarchical synthesis improve scalability?

Since monolithic synthesis does not scale, we attempt hierarchical synthesis (§4.7) to generate a model for the load_store_unit. Hierarchical synthesis is feasible since the two requirements outlined in §4.7 are satisfied: (a) signals of the store_unit are not directly written to by outside logic and (b) almost all inputs to the top-level load_store_unit are passed to the (sub-component) store_unit. The exception to (b) is an input signaling whether the incoming request into the load_store_unit is a load/store. We conditioned it to be a store before passing it to the store_unit. By adopting the previously (monolithically) synthesized store_unit, we only needed to synthesize the non-store_unit guards. While monolithic synthesis timed out, we could generate the model using the hierarchical approach in ~12m (★ in Tab. 3).

Hierarchical synthesis is generally suitable when the composition is performed at module boundaries of the RTL. Some cases tend to satisfy condition (a) from §4.7. However, as seen in this case, condition (b) from §4.7 may be harder to satisfy if there is intermediate logic between the trigger signals of the parent module and those of submodules. While the logic was manually identifiable in this case, in the future one could use a version of guard synthesis to extract it. Thus hierarchical lifting leads to better scalability when synthesizing complex models. However, this might require the user to provide additional instrumentation/hints.

5.3.3 Is performance of security analysis better with the micro-update model compared to source RTL?

**Experimental setup** We investigate whether using the lifted model can improve the performance of security verification. Since we only lifted a model corresponding to the load_store_unit, we first wrap the generated model in a simple processor shim. This shim executes alui, lw, and sw instructions. While alui instructions are executed locally, the shim interfaces with the load_store_unit for sw and lw instructions. Execution is stalled if the load_store_unit is busy. Developing this shim required around 5-6 person-hours of work. We also wrap the source RTL

| Model slice | Signals-of-interest ($S_{data}$) | $|S_{data}|$ | $|L|$ | Simulation examples | Distinguish/cover examples | Guard synthesis | Equivalence check (d = 15) |
|-------------|----------------------------------|------------|-------|---------------------|--------------------------|----------------|---------------------------|
| I-type ALU  | $S_1 = $ decode, register file I/O, ALU, bypassing | 15 | 25 | 190 | 1 | 1m1s | 10m9s |
| R-type ALU  | $S_1 = $ decode, register file I/O, ALU, bypassing | 15 | 29 | 190 | 3 | 2m4s | 11m57s |
| ALU + lw + sw | $S_2 = S_1 \cup$ memory ports, LB | 20 | 37 | 190 | 8 | 57m | 34m24s |

**Table 1.** Model parameters, example counts, synthesis and equivalence check times for the generated models for Sodor5Stage.

<table>
<thead>
<tr>
<th>Symbolic instruction sequence (testcase)</th>
<th>Source signals (sv)</th>
<th>Outcome</th>
<th>Design runtime</th>
<th>Model runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop lw</td>
<td>mem, regs</td>
<td>Safe</td>
<td>2m4s</td>
<td>8s</td>
</tr>
<tr>
<td>nop lw</td>
<td>mem, regs</td>
<td>Unsafe</td>
<td>3m50s</td>
<td>11s</td>
</tr>
<tr>
<td>nop alui lw</td>
<td>mem</td>
<td>Safe</td>
<td>2m37s</td>
<td>12s</td>
</tr>
<tr>
<td>nop sw lw</td>
<td>mem</td>
<td>Safe</td>
<td>2m30s</td>
<td>9s</td>
</tr>
<tr>
<td>nop lw lw</td>
<td>regs</td>
<td>Unsafe</td>
<td>4m15s</td>
<td>8m51s</td>
</tr>
<tr>
<td>nop( lw + alui)∗ sw</td>
<td>mem, regs</td>
<td>Safe</td>
<td>4m49s</td>
<td>38s</td>
</tr>
<tr>
<td>nop( lw + alui)³ sw</td>
<td>mem, regs</td>
<td>Safe</td>
<td>5m11s</td>
<td>1m31s</td>
</tr>
</tbody>
</table>

**Table 2.** Comparison of information-flow proof run times over some representative testcases. Tests are over symbolic sequences of instructions represented as a regular expression in the first column. The source run time is over the Sodor5Stage RTL (with wbuffer) while model run time is over our lifted ALU/LS model.
for the `load_store_unit` in an identical shim for a valid comparison. We now conduct an experiment to compare security analysis run times when using the (shim-wrapped) lifted model/source `load_store_unit` RTL as the underlying HW platform (similar to §5.2.1).

### The `load_store_unit` timing channel and results
In our experiment, we analyze whether a given software instruction sequence is vulnerable to a `load_store_unit`-based timing channel. We formulate invulnerability as a non-interference [21] property stating that the timing behaviour of the instruction sequence is independent of victim data (we assume that the victim’s secret resides in data memory). We check this property by invoking the SymbiYosys model checker. If the check passes, the instruction sequence is secure while a counterexample would indicate a vulnerability.

We present our results in Tab. 4, where rows denote the symbolic test cases that are checked. The test cases are based on read gadgets seen in hardware attacks. We observe that verification run times with the lifted model are up to 8× lower than the source RTL. This tends to increase for larger tests.

We also note that certain test programs are unsafe (US). This is the case since the `load_store_unit` blocks loads when there are previous pending stores at the same address. Thus, the timing behaviour of `sw · lw` when the `lw` and `sw` are on the same address is different than when they are on different addresses. If the address constitutes a victim secret, an attacker could infer the secret through a timing-based attack [25].

This demonstrates that models lifted from security-relevant design components can be used to check the existence of vulnerabilities in SW with significant performance improvement.

### Evaluation highlights
Our experiments demonstrate: (a) the feasibility of micro-update model lifting, and the ability to improve scalability through hierarchical synthesis, and (b) the application of the lifted models to perform security analysis of software with greater reliability than handwritten models and better performance than with source RTL.

## 6 Discussion and Limitations

### Manual effort
The main manual effort required in our approach is for identifying signals-of-interest ($S_{data}$), the micro-update library (L) and design instrumentation. The first is fundamental to our framework as $S_{data}$ captures the design slice the user is interested in analyzing. While the current approach requires a user-specified micro-update library, going forward we foresee automating this by utilizing techniques such as specification/invariant-mining [22, 45]. As non-design-experts, a large fraction of the manual effort in our case studies, (∼ 3-4 person weeks), was spent in understanding and instrumenting the RTL designs. Comparatively, the time required to write the actual micro-update library and the processor shim (in §4) was much smaller, around 1-2 person days and 4-5 person hours respectively. We believe that if lifting is performed in lockstep with the design phase, and with aid from RTL designers, the time required can be reduced.

### Signals-of-interest and analysis coverage
While we require the user to identify the signals-of-interest, automatically identifying the complete attack surface is a major open challenge. Prior work on analyzing hardware exploits also rely on manually identified signals (e.g. [18, 29, 30, 55, 70]). Through the signal-of-interest ($S_{data}$), our approach exposes a tradeoff between model coverage and scalability. While a model capturing with smaller $S_{data}$ is less detailed, it can still be used to prove security against multiple attack scenarios targeting those signals (e.g. in §5.3.3 we study security implications of a `load_store_unit` against several instruction sequences). For generating a model with high coverage, hierarchical synthesis (§4.7) can improve scalability.

## 7 Related Work
Several approaches propose formal specifications/models at the ISA-level, such as the RISC-V specification [75], as well as formal models of ISA-level semantics (e.g. in SAIL [28] and Kami [41]). Approaches such as instruction-level-abstraction (ILA) [36, 37, 81, 83] extend these models to include additional architectural state (e.g. accelerator state). ILA-MCM [83] also considers a model which is
composed of code blocks, however, these are composed axiomatically. These models have been extensively used for specification [7] and verification of processor designs (e.g. [14, 61, 66, 74]). There is also work on synthesizing such models from hardware [68]. However, as discussed in §2.3, these approaches are not precise enough to model microarchitecture-level interactions, which is essential for accurate security analysis.

There is work on developing microarchitectural models of hardware, such as \( \mu \text{spec} \) [49]. \( \mu \text{spec} \) enables modelling of model memory consistency [48], coherence [52], and security properties [70] with modular and hierarchical verification [50]. More recently, there has been work on generating \( \mu \text{spec} \) models from RTL [35] as well as work on automatically converting axiomatic \( \mu \text{spec} \) models to equivalent operational models [26], [55] uses a \( \mu \text{spec} \) based model to detect vulnerabilities. As discussed in §2.4, axiomatic modelling and the absence of functional detail makes checking equivalence with RTL and performing deeper specification (as required in microarchitectural security analysis) challenging.

More broadly, our work is related to program slicing from the software engineering domain (e.g., [34, 76]). Given a set of program variables, program slicing identifies a sub-program (slice) that affects the behaviours of these variables. There is also work that also performs slicing for hardware [82], where the slice captures architectural signals instead of program variables. We, however, allow the generation of models with microarchitectural signals Additionally, program slicing relies on an (often syntactic) data/control-flow analysis to identify the slice as a subset of the existing program. Our model, on the other hand, replaces the original RTL circuit with micro-updates from the user-provided library, leading to smaller and more abstract models.

Approaches that define [31] and verify [18, 23, 29, 30] security properties from a software standpoint manually develop platform models for program execution. This is error-prone due to subtle microarchitectural interactions. In §5.2.1 we demonstrated how micro-update models generated with strong guarantees can increase the level of assurance of these approaches. [80] develop a speculative platform model with a guarantee of invulnerability to some attacks, while [13] develops a methodology to validate platform models.

Unique Program Execution Checking (UPEC) [24] performs verification of RTL by checking whether executing read gadgets (e.g. [42, 44]) leads to a security vulnerability. This is orthogonal since we lift models from RTL for scalable software verification. In particular, a UPEC-like approach could be soundly performed on the lifted model owing to our equivalence guarantee (§4.2). Moreover, as our results indicate, verification against the lifted model is more performant as compared to the RTL.

Micro-updates bear resemblance to rules from BlueSpec [57] or transactions from TLM [15]. However, our focus is modelling and lifting as opposed to hardware design.

8 Conclusion

In this work, we proposed micro-update models as a formalism for developing abstract formal models of the microarchitecture. By accurately preserving security-relevant microarchitectural detail, micro-update models provide an abstract yet sound substrate for performing scalable security verification of software. To address the challenge in hand-writing formal models, we developed a semi-automated technique to hierarchically synthesize micro-update models from RTL. We evaluated our approach by synthesizing models from the Sodor5Stage and cva6 processors. We demonstrated how the generated models can be used to soundly perform semantic security analysis, with improved performance over the source RTL.

Our modelling and lifting framework can allow HW designers to increase the level of assurance of security analysis while reducing the efforts involved in formal model development.

Acknowledgements

We thank our reviewers and our shepherd Kristin Barber for their comments and feedback which helped improve this paper. This work benefitted from discussions with Kevin Laeufer, Federico Mora, Jonathan Shi from UC Berkeley, and Scott Constable and John Matthews from Intel Labs. This work was supported in part by Intel Corporation under the Scalable Assurance program, DARPA contract FA8750-20-C0156 and NSF grant 1837132.

References


